(4)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-397

TEMPORAL, PROCESSOR, AND SPATIAL LOCALITY IN MULTIPROCESSOR MEMORY REFERENCES

Anant Agarwal
Anoop Gupta

June 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-397	5. MONITORING ORGANIZATION REPORT NUMBER(S) N0014-87-K-0825
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO
11. TITLE (Include Security Classification) Temporal, Processor, and Spatial Locality in Multiprocessor Memory References	
12. PERSONAL AUTHOR(S) Agarwal, A. and Gupta, A.	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1989 June 15. PAGE COUNT 18
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB-GROUP	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Locality, temporal locality, spatial locality, cache coherence, shared-memory multiprocessors, multiprocessor locality, network traffic, snoopy caches

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The performance of cache-coherent multiprocessors is strongly influenced by locality in the memory reference behavior of parallel applications. While the notions of temporal and spatial locality in uniprocessor memory references are well understood, the corresponding notions of locality in multiprocessors and their impact on multiprocessor cache behavior are not clear. A locality model suitable for multiprocessor cache evaluation is derived by viewing memory references as streams of processor identifiers directed at specific cache/memory blocks. This viewpoint differs from the traditional uniprocessor approach that uses streams of addresses to different blocks emanating form specific processors. Our view is based on the intuition that cache coherence traffic in multiprocessor is largely determined by the number of processors accessing a location, the frequency with which they access the location, and the sequence in which their accesses occur. The specific locations accessed by each processor, the time order of access to different locations, and the size of the working set play a smaller role in determining the cache coherence traffic, →

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985—607-047

Unclassified

19. although they still influence intrinsic cache performance. Looking at traces from the viewpoint of a memory block leads to a new notion of reference locality for multiprocessors, called processor locality. In this paper, we study the temporal, spatial, and processor locality in the memory reference patterns of three parallel applications. Based on the observed locality, we then reflect on the expected cache behavior of the three applications.

Temporal, Processor, and Spatial Locality in Multiprocessor Memory References*

Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Anoop Gupta
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

The performance of cache-coherent multiprocessors is strongly influenced by locality in the memory reference behavior of parallel applications. While the notions of temporal and spatial locality in uniprocessor memory references are well understood, the corresponding notions of locality in multiprocessors and their impact on multiprocessor cache behavior are not clear. A locality model suitable for multiprocessor cache evaluation is derived by viewing memory references as streams of processor identifiers directed at specific cache/memory blocks. This viewpoint differs from the traditional uniprocessor approach that uses streams of addresses to different blocks emanating from specific processors. Our view is based on the intuition that cache coherence traffic in multiprocessors is largely determined by the number of processors accessing a location, the frequency with which they access the location, and the sequence in which their accesses occur. The specific locations accessed by each processor, the time order of access to different locations, and the size of the working set play a smaller role in determining the cache coherence traffic, although they still influence intrinsic cache performance. Looking at traces from the viewpoint of a memory block leads to a new notion of reference locality for multiprocessors, called processor locality. In this paper, we study the temporal, spatial, and processor locality in the memory reference patterns of three parallel applications. Based on the observed locality, we then reflect on the expected cache behavior of the three applications.

1 Introduction

Multiprocessors often use caches to reduce their network bandwidth requirements. Caches retain recently accessed data so that repeat references to this data in the near future and will not require network traversals. Repeated access to the same data in a given interval of time is the property of temporal locality of memory references and has been well studied in single processor systems [1, 2]. Spatial locality of memory references is another related property of memory references that places a high probability of access to data close to previously accessed data. Again, this property of single processor programs has been widely observed. The viability of cache-coherent multiprocessors is strongly predicated on whether the multiprocessor caches can exploit locality of memory referencing.

*Preliminary results of this study were reported in Sigmetrics 1988.

Clearly, a thorough understanding of the memory access patterns of parallel processing applications is necessary to determine a suitable organization of the memory hierarchy in multiprocessors. For example, several cache consistency algorithms proposed in the literature are based on subtle differences in the expected memory reference patterns; lacking a characterization of multiprocessor memory referencing locality, it is hard to obtain insight into the benefits of one scheme over another. While some previous studies have looked at shared-memory reference patterns (e.g., [3]), they did not analyze the temporal, spatial, and processor locality of shared data.

Unfortunately, multiprocessor locality models that we can use to aid in our understanding of the reference patterns of parallel systems do not exist. The well known notions of locality in single processor systems do not carry over straightforwardly. Consider, for example, the sequence of memory references $r_1r_2r_3r_4r_5r_6$ to the same memory block. While such temporal locality can be usefully exploited by a uniprocessor cache, the degree to which a multiprocessor uses such locality depends on which processor made the individual references and whether the references were reads or writes. The negative extreme case would correspond to each reference being a write and emanating from a different processor.

Similarly, block size effects are hard to estimate. Increasing the block size could improve useful locality by capturing additional data words in the block that will be referenced by the processor in the near future. However, two data words being written by different processors could fall into the same block owing to a block size increase and prove harmful to cache performance.

We present a simple characterization of multiprocessor memory references and derive a locality model that is useful in a multiprocessor context. The key to the model is that we focus on the set of references by one or more processors to a *given* memory block. We introduce the notion of *processor locality* as the average number of repeat references to a memory block by the same processor. Specific variations of processor locality can be defined for use in different applications. For example, one interesting form of processor locality that provides insight into ownership-based cache coherence schemes is the sequences of repeat references to a given memory block by the same processor, given that at least one of the references is a write [4]. A slightly different definition might count just the number of writes to a block by the same processor before a reference by another processor. Eggers and Katz [5] proposed using such a metric in characterizing multiprocessor memory references.

Besides its obvious use in gaining insight into the performance of cache coherence schemes, processor locality metrics can also be used to evaluate the efficacy of block structuring algorithms proposed to enhance locality in memory referencing of shared memory multiprocessors.

We use our locality characterization to analyze the locality patterns in three parallel applications using address trace data. Multiprocessor address traces are derived from these parallel applications running under the MACH operating system on a shared-memory multiprocessor. An extended ATUM address tracing scheme implemented on a 4-CPU DEC VAX 8350 [6] provided the trace data used in this study. The applications include ParaOPS5—a parallel implementation of the OPS5 rule-based language, P-Thor—a parallel logic simulator, and LocusRoute—a global router for VLSI standard cells.[1]

Our results suggest that shared references display a significant amount of temporal locality and only a moderate amount of processor locality. The average number of read and write

[1]Note that these programs were called POPS, THOR, and PEROUTE in our original Sigmetrics '88 paper [4]. We have renamed them here to be consistent with other recent papers [5] that deal with the same applications and as desired by the authors of the applications.

references to a write-shared block before a remote reference are 4 and 2 respectively. This locality is exploited by the write-back class of cache coherence schemes to reduce the cost of references to shared data.

This paper is organized as follows. Section 2 defines our multiprocessor model and the terminology used throughout the paper. Section 3 presents background information about the ATUM address tracing technique and the applications measured. Sections 4 constitutes the bulk of the paper and is devoted to analyzing locality in the parallel traces, and studying the impact of the reference characteristics on cache consistency algorithms. Specifically, Section 4.1 assesses the temporal locality in shared references, Section 4.2 the processor locality, and Section 4.3 analyzes spatial locality in the traces. Section 4.4 focuses on how the memory reference characteristics affect the performance of various cache consistency algorithms. Section 5 concludes the paper.

2 Characterization of Memory References

This section presents the multiprocessor model and introduces some nomenclature to help explain memory access patterns in multiprocessors. The notion of processor locality is also introduced.

2.1 Multiprocessor Model and Definitions

The multiprocessor model we assume for our analyses is straightforward. We assume that the system consists of several processors each with its own cache memory. Memory is accessed through an interconnection network. We make the simplifying assumption that caches are infinite in size to concentrate on traffic caused owing to cache coherence related actions. The specific organization of the network and memory system is, however, unimportant to our characterization of locality.

We first introduce some nomenclature to help explain memory access patterns. A *block* is the unit of data transfer between the cache and main memory. The block size is assumed to be 1 word (4 bytes) unless otherwise stated. The small block size is chosen so that the reference behavior for each data object can be derived. However, characterization using larger block sizes is also important to study the spatial locality of shared objects, and is dealt with in Sections 4.3 and 4.4. A *read-shared* block is one that is shared (accessed by multiple processors), but never written into for the duration of the trace. A *write-shared* block is one that is shared, and written at least once. A *cpu-shared* block is one that is either read shared or write shared.

It is useful to have a notion of time in the context of multiprocessor execution. Our traces contain interleaved memory accesses by the various processors in approximately the same order they occurred. However, the exact time at which the reference was made is not clear. For example, if the processors i, j, and k each made references at real time instants t, $t+1$, and so on, the trace might have the references $i_t, j_t, k_t, j_{t+1}, i_{t+1}, k_{t+1}$, where the order of the t^{th} references of the 3 processors might be random with respect to each other. The traces also show clusters of memory references by the same processor, and the time interval between references by the same processor also varies.

Owing to such statistical variations in the reference pattern, we will use an approximation to real time. The order of occurrence of a reference in the trace is our index of time. So the r^{th}

reference in the trace is considered to have occurred at time r.[2] Because the paper considers several cases where the traces are filtered to extract specific references (e.g., shared user data), to enable comparisons, the time index used for a reference depends on its index in the original trace. For example, when we filter out operating system references while studying sharing in the user address space, the time index of a user reference corresponds to its position in the unfiltered trace.

The ensuing definitions for displaying multiprocessor locality focus on the *sequence of processors* referencing a *given memory block*. Contrast this viewpoint with uniprocessor locality that typically focuses on the *sequence of memory addresses* referenced by a *given processor*. A reference to a block B by processor i is said to *ping* if the previous reference to that block was by processor j, where $j \neq i$. We call such a reference a *pinging reference*. Conversely, a reference to a block B by processor i is said to *cling* if the previous reference to that block was also by processor i. Such a reference is called a *clinging reference*. By these definitions, a ping can only occur on a reference to a shared block. Pings and clings to a block are determined simply by keeping track of which processor last referenced a block. Similarly, the current state of a block, clean or dirty, is determined solely by the references of the processor accessing it currently. A block is said to be dirty if it has been written into since the previous pinging reference to it. Therefore, a block always starts out clean following a pinging reference to it.

Figure 1 depicts read/write references to a given memory block, where the number in the second row corresponds to the processor accessing the block. The reference by processor 3 at time $t+23$ is a pinging read reference, the reference at time $t+25$ is a clinging write reference.

2.2 Characterizing Locality

The notion of clings and pings allows the derivation of simple criteria for multiprocessor memory reference locality. The appealing feature of clings and pings is that they do not depend on implementation details such as cache sizes. In addition, they provide useful information about cache performance. For example, assuming a local cache, clinging read references do not cause a network transaction; on the other hand, pinging write reference always cause a network transaction. The ensuing discussion uses statistics derived from pings and clings to study locality.

Temporal locality is displayed by references to a given block of data that are clustered in time. Small time intervals between clinging references denote a useful form of temporal locality in multiprocessors; conversely, small time intervals between pinging references is potentially harmful. In the reference sequence depicted in Figure 1 temporal locality of clinging references is more evident.

Time intervals between pinging and clinging references are a useful method of depicting the temporal locality of shared-memory references and can yield useful insights into the behavior of small caches in multiprocessor environments. However, a block might reside in a large cache for long periods of time without being displaced, making the relative sequence of references to a given block by various processors a more important determinant of cache performance. The form of locality that becomes more important, then, is called processor locality.

Processor locality is the tendency of a processor to access a block repeatedly before an access

[2] We believe that fine time distinctions are not significant in our study. To approximate real time, one can keep a virtual system time incremented by one unit for every n references in the trace where n is the number of processors. In other words, the times specified in our paper can be divided by 4 to get a rough idea of the real time.

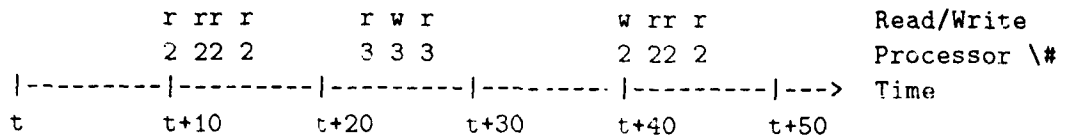

Figure 1: Characterizing locality in multiprocessor memory references. Various processor accesses (represented by the numbes in the second row) of a given block B are shown. r/w stand for reads or writes. The time instants with no corresponding references imply accesses of blocks other than B.

from another processor, and is measured by the average length of the sequences of clinging references. In Figure 1, the average number of clinging references before a pinging reference is $(4+3+4)/3$.

We can derive a class of processor locality metrics for use in different applications. For example, a characterization that does not distinguish between read and write references is enough to analyze cache coherence schemes such as the Dir_1NB directory scheme studied in [8]. However, this definition is unsuited for a cache coherence scheme that allows multiple cached copies of clean blocks. Therefore, a more practical definition of processor locality measures *the average length of those sequences of clinging references, where at least one reference is a write*. This definition yields $(3+4)/2$ as the measure of processor locality for our example.

In general, we can use the following notation to describe a processor locality metric: $r_c^{+/*}\ w_c^{+/*}\ t_p$. Here, r and w denote reads and writes to a block by a given processor, $+$ denotes one or more, and $*$ denotes zero or more. Sequences by the same processor are terminated by a pinging reference of type t. The type of the pinging reference can be a *read*, *write*, or *either* (denoted r, w, r/w). The length of the $r_c^{+/*}\ w_c^{+/*}$ sequence determines the processor locality. In this notation, the two definitions of processor locality in the previous paragraph are $r_c^*\ w_c^*\ r/w_p$ and $r_c^*\ w_c^+\ r/w_p$ respectively.

Processor locality measures locality in shared references alone. It is meant as an aid to gain insight into the shared reference patterns of parallel programs and usually cannot be used to obtain performance data directly. For instance, an application that has very few shared references will have a low rate of cache coherency related transactions even with abysmal processor locality. Consequently, a performance model might consider using the fraction of shared references in addition to the processor locality parameter.

A direct impact of processor locality is noticed in the performance of various cache consistency schemes, which exploit different locality patterns in references to read-shared or write-shared blocks. Notice that a high temporal locality of pinging references yields a low processor locality, and negatively impacts the performance of multiprocessor caches.

Spatial locality is the tendency of processors to access data in the vicinity of a recently accessed memory word in a given interval of time. Clearly, a useful form of spatial locality increases the probability that a given processor accesses words in the neighborhood of words it accessed recently, while the opposite form of spatial locality increases the rate at which *other* processors access these words. Put another way, spatial locality can be useful in multiprocessors if a larger block size increases the processor locality of shared references. As we will show in Section 4.3 increasing the block size does not always increase the processor locality.

3 Applications and Data Collection

Our study is based on trace analysis. The traces are obtained using a multiprocessor extension of the ATUM tracing scheme [9]. ATUM stands for Address Tracing Using Microcode and works as follows: During the execution of each instruction, the microcode writes out the memory references made by the processor to a portion of memory reserved for tracing. In the multiprocessor extension of ATUM, each access to trace memory is interlocked to enable the microcode in several processors to write their references to this memory. Thus a trace contains interleaved address streams of several processors. The traces used for this study were gathered on a 4-CPU VAX 8350 machine running the MACH operating system. Each trace is roughly 3.5 million references long. In addition to addresses, ATUM records the opcodes, and the virtual-to-physical translations that occur during translation-lookaside-buffer misses. A location is considered shared when it is referenced by more than one CPU. Because different processes could access a given shared location with different virtual addresses, sharing is detected by translating the various virtual addresses of a shared location to its common physical address.

The traces used in this paper are obtained from three programs: ParaOPS5, P-Thor, and LocusRoute. ParaOPS5 [10] is a parallel implementation of a rule-based programming language called OPS5, which is a widely used languages for the building expert systems. It exploits parallelism at a fine granularity and makes extensive use of the shared memory provided by the architecture. P-Thor is a parallel implementation of a logic simulator done by Larry Soule at Stanford University. The simulator transforms the task of circuit simulation into a series of node evaluations, where each node corresponds to a device in the circuit. The parallel implementation evaluates these nodes in parallel, while handling the dependencies between them. **LocusRoute**, is a parallel VLSI router written by Jonathan Rose at Stanford [11].

3.1 General Statistics

Tables 1 and 2 present some trace statistics relevant to this study. Because the instruction space is usually read-only, it can be treated specially in memory management, and so the statistics presented in this paper correspond to data references alone. The columns in Table 1 denote the total number of user references, user data references, user data shared, and shared write references. Instruction and data references are about equal as expected. In ParaOPS5, P-Thor, and LocusRoute, shared data references comprise roughly 20%, 10%, and 3% of all user references. The corresponding fractions of shared write references are about 3%, 1%, and 0.2%.

Table 1: Summary of dynamic trace characteristics.

Trace	User References (thousands)	Data References (thousands)	Shared References (thousands)	Shared Writes (thousands)
ParaOPS5	2817	1346	576	77
P-Thor	2727	1527	326	24
LocusRoute	3242	1528	119	6

The statistics in Table 2 display the number of unique user blocks, unique shared blocks, and the unique shared written blocks in the traces.

Our analyses in this paper focuses on user references alone. Except P-Thor, our applications did not have a significant amount of process migration related sharing: the few blocks that are

Table 2: Summary of static trace characteristics. Only user data blocks are considered

Trace	Data Blocks (thousands)	Shared Data Blocks (thousands)	Write-Shared Blocks (thousands)
ParaOPS5	29.3	19.8	4.0
P-Thor	71.9	1.8	1.3
LocusRoute	11.6	3.3	0.7

shared by multiple processors solely due to process migration are not counted in with shared blocks. Results on sharing in the operating system, and sharing owing to process migration can be found in [4].

4 Results and Analyses

This section first analyzes temporal locality in the traces. We then evaluate the processor locality in the traces and the impact of block size on this parameter. We evaluated three different cache coherence schemes by the amount of traffic they generate for various block sizes. This paper summarizes our findings and uses processor locality as a means of gaining insight into their behavior. Unless stated otherwise, we assume infinite caches and 4-byte blocks.

4.1 Temporal Locality

This section deals with dynamic memory access patterns and characterizes the temporal locality of cpu-shared user data references. We present the median of the distribution of time intervals between clinging and pinging references in Table 3 to demonstrate the temporal locality of data references.[3] The average interval of time between accesses to the same shared block tends to be large because even one reference with a very large interval (or an *outlier*) can skew the average towards large values. Such outliers are not important for two reasons. First, in practical finite sized caches the much shorter cache lifetime of blocks would preclude such large values. Second, the large values in our applications is chiefly due to clinging references that occur when the process resumes execution on the same processor after being switched out. Therefore, in the context of time intervals, a more interesting number is the median, or the time interval over which half the clinging or pinging references occur.

Table 3: Temporal locality characteristics. Only user data blocks are considered. Block size is 1 word (4 bytes) Numbers denote the median of the frequency distribution of time intervals between three events: clinging references, pinging references, and pinging references to a dirty block.

Trace	Cling References	Ping References	Pings to Dirty Blks
ParaOPS5	23	10	363
P-Thor	25	7	1779
LocusRoute	28188	13869	19711

In ParaOPS5 and P-Thor over 50% of the intervals between clinging references are 25 time units or less. Not surprisingly, these numbers show that blocks are re-referenced at small intervals

[3] For detailed frequency distribution graphs, see [4].

of time—which is simply a reconfirmation of the belief that memory references display a high temporal locality, and is the precise reason caching is successful.

LocusRoute has a much larger interval. In LocusRoute, wires are selected at random and a route is chosen using cost values from a shared matrix. Because a wire might be selected by a processor at random, there is no significant temporal locality in referencing the elements in the cost matrix. An algorithm with better temporal locality might favor routing wires in a given neighborhood rather than choosing a wire at random to increase the probability a given word is re-referenced soon. Such a choice will benefit spatial locality also.

These temporal locality results are compared with those for pinging references, or for a reference to a block by a processor followed by a reference from another processor. The time intervals here are interestingly lower than for clinging references, which says that references to shared blocks by different processors are usually at least as finely interleaved as references by the same processor. Doubtlessly, the cause of the high temporal locality of pinging references is that our applications exploit parallelism at a fine granularity, and the use of spin locks for synchronization.

As an interesting aside, in addition to a first peak at a low time interval, our frequency distribution for pings showed a small second peak at 256 time units in P-Thor owing to the process migrating to another processor following a context switch. If the level of process migration is high, this peak at a large time interval can become much taller, which falsely suggests that process migration lowers the temporal locality of shared references. In reality, process migration simply makes a large fraction of the logically private blocks appear shared, and it is references to these shared blocks alone that causes the tall second peak.

The previous results did not distinguish between read and write references. Making this distinction is necessary because in many high-performance multiprocessor architectures, writes and pinging references to dirty blocks cause bus traffic because the new value of the dirty block must be somehow transmitted to the requesting processor. The time interval between pinging references to a dirty block for the three applications is far greater than the corresponding time between *all* pinging references. The high frequency of pinging references at low time intervals is therefore attributable to read references. A possible case is the test-and-test&set synchronization sequence, where one might expect multiple reads from several processors, but less frequent writes. The low temporal locality in pinging references to dirty blocks encourages us to believe that for large time periods blocks can be considered as private and no traffic need be generated in maintaining consistent caches. One conclusion of this observation is that cache management schemes must support efficient read sharing of blocks.

4.2 Processor Locality

As caches grow bigger, blocks are expected to stay in the cache for long periods of time. Then, a better characterization uses the notion of processor locality. Our discussion here addresses processor locality in two ways. The first uses the number of clinging references to a block ($r^* w^* r/w_p$), and the second the number of clinging references to a block, given that at least one of the references was a write $r_r^* w_r^+ r/w_p$.

Figure 2 shows the frequency histogram of the number of clinging references to a block, given at least one reference was a write. Due to the wide range of the number of references, the bins on the X axis increase in powers of two: a bar at x with height y in the frequency histogram plot implies y sequences of clinging references of length t, such that $x \leq t < 2x$. Here, we will use

averages because the average is more indicative of processor locality than the median; outliers represent a large number of references, and must be weighted accordingly.

Several observations can be made from Figure 2. First, the average number of clinging references to written blocks is 5.6 for ParaOPS5, 3.6 for P-Thor, and 7.5 for LocusRoute. Write references are much fewer than reads and contribute 1.6, 1.7, and 1.2 respectively to these averages. The write reference sequences correspond to the form of processor locality denoted $w^+ r/w_p$.

We found a significantly lower processor locality in the distributions for clinging references when we relaxed the requirement that each sequence have at least one write [4]. For example, for P-Thor, there are about 200,000 pinging references to a block referenced only once by the previous processor. The correspondingly low average of 1.3 for P-Thor indicates that interleaved references by different processors are as frequent as clinging references, implying low processor locality. (The averages for ParaOPS5 and LocusRoute were 1.8, and 2.5 respectively.) A cache consistency scheme that allowed only one cached copy of any block [8] performed abysmally for this very reason. Another important observation is that the total number of pinging references to dirty blocks are approximately an order of magnitude lower than all pinging references, which lowers the overall rate of cache consistency related transactions.

One of the chief differences between some of the cache consistency schemes is the way they treat write references. One set of schemes, e.g., DRAGON [12] or FIREFLY [13], allow caches to hold valid copies of blocks that are being written into by others, and receive updates of the values on writes. Another set of schemes allow only one copy of a written block (e.g., Berkeley Ownership [14], or various flavors of directory schemes [8]). The performance of update versus invalidate is predicated on the locality of references to write-shared blocks. As noted earlier, the average number of writes to a block before a pinging reference is small although not unity (1.7 for P-Thor) implying that either method will not overwhelmingly outperform the other. Our results in the next section show that the invalidate and update schemes perform similarly for 1 word block sizes and bear out this intuition.

There are several possible reasons for the low value of clinging write references. We expect a low value for write references to spinlocks. We also expect this value to be low for migratory shared objects [7] which move from one processor to another, with each processor making some modifications to the object. Also mostly-read-only objects are written once, and then numerous pinging read references are made by other processors.

We also studied the distribution of the number of clinging write references. A surprising observation was that a significant fraction of clinging sequences had exactly one write. The larger average is due to a small number of clinging write sequences with several tens of writes. This dichotomous nature of clinging sequences suggests that competitive cache coherence schemes [15] that can resort to invalidations when the number of write updates crosses a threshold might be the right scheme to use. We also noticed that it was usually the synchronization objects that resulted in a clinging sequence with exactly one write. So another possibility would be to use an update-based protocol for synchronization objects, while using an invalidation-based coherence protocol for all other data objects. In an environment where processes can migrate, yet another scheme might use invalidations for private data objects spuriously shared due to process migration and use updates for other blocks.

In summary, we saw that the processor locality of shared-references is moderate, with roughly 2 writes and 4 reads on average to write-shared objects before a pinging reference. Given the moderate processor locality of shared-data, invalidating schemes such as the Berkeley Owner-

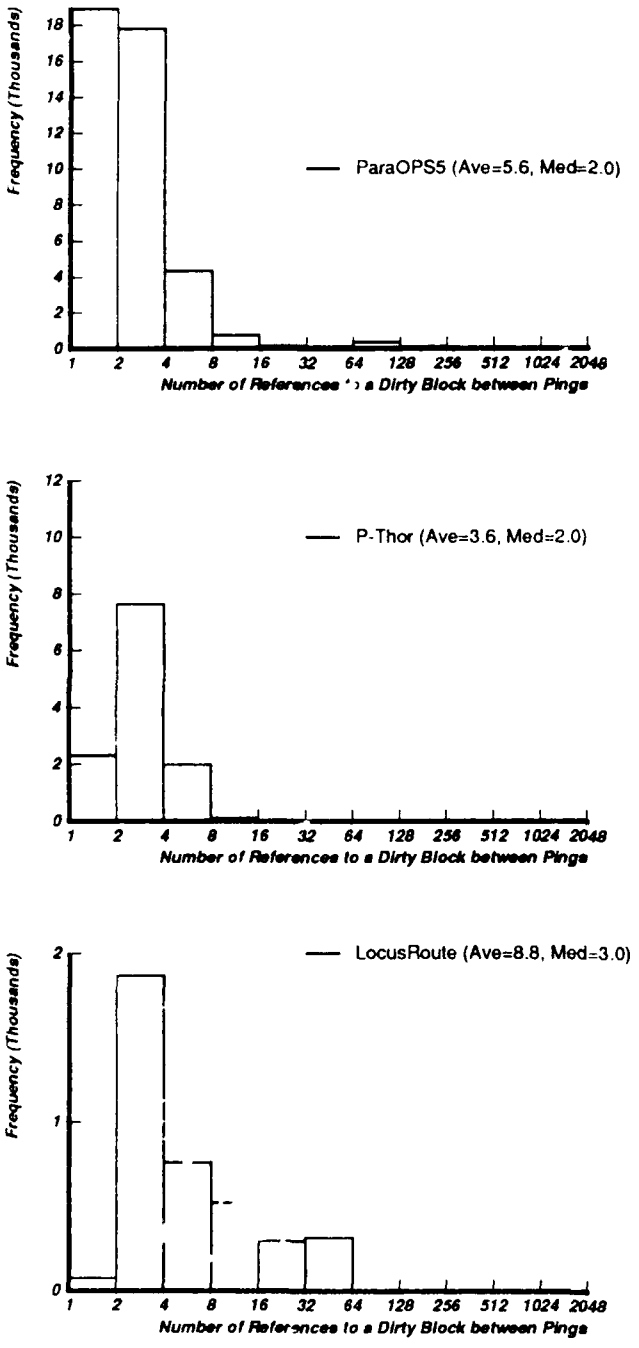

Figure 2: Distribution of the number of references to a block before a pinging reference to the same block, given that at least one reference was a write. Only *shared data* references of *user* are included.

ship protocol or directory schemes, and the updating protocols such as the Dragon and Firefly schemes, are expected to have similar performance. We verified this using simulation in Section 4.4.

4.3 Spatial Locality

We now examine the effects of spatial locality on the performance of cache coherence schemes. Figures 3 and 4 plot processor locality histograms for the three applications for block sizes of 16 and 64 bytes. The averages for the three block sizes are shown in Table 4.

Table 4: Spatial locality and the impact of block size. Only user data blocks are considered. Numbers denote the average of the number of clinging references to a block, at least one of which was a write.

	Block Size		
Trace	4 bytes	16 bytes	64 bytes
ParaOPS5	5.6	7.8	7.6
P-Thor	3.6	5.1	6.2
LocusRoute	8.8	29.2	117.2

Increasing the block size impacts the various applications differently. LocusRoute shows a substantial improvement in processor locality (8.8 to 117.2) as block size is increased from 4 to 64 bytes. The reason for the substantial improvement for LocusRoute is that it has a central data structure, called the *cost array*, accessed very frequently and in a regular fashion, thus resulting in high spatial locality of references.[4] In comparison to LocusRoute, both P-Thor and ParaOPS5 show improvements of a much smaller magnitude, and in fact, the processor locality measure decreases slightly for ParaOPS5 as we go from 16 to 64 byte blocks.

Why does block size impact processor locality so differently for various shared applications? As the block size is increased the potential for references to adjacent words increases and two opposing forces come into play. If the probability a given processor accesses a word in the vicinity of a word it accessed before increases, then the processor locality is likely to improve. Contrarily, a larger block size increases the probability of unrelated shared words residing in the same block, and a write to one word can cause a ping to the entire block currently being accessed by another processor. Clearly, the applications display differing degrees of both effects.

Let us lock at the issue of same processor versus different processor accesses of a block more concretely. Examine the processor locality distributions for ParaOPS5 when block size is 16 bytes and when it is 64 bytes (see top of Figures 3 and 4). We see a significantly larger number of occurrences with 8-16, 16-32, 32-64, and 64-128 clings before a ping as we move from 16 to 64 byte blocks. This increase is due to the spatial locality in the references of a single processor, the positive force. However, we also see a significant increase in the number of occurrences where *there are only 1-2 clings before a ping* as we move from 16 to 64 byte blocks. This is a result of the interference effect discussed above, and overall it nullifies the advantage of the large block size.

[4] Note that the height of the distribution becomes smaller as block sizes are increased because even small values at the tail end of the distribution correspond to a large number of references. For example, in LocusRoute, there are 18 sequences of length between 256 and 512, which account for several thousand references.

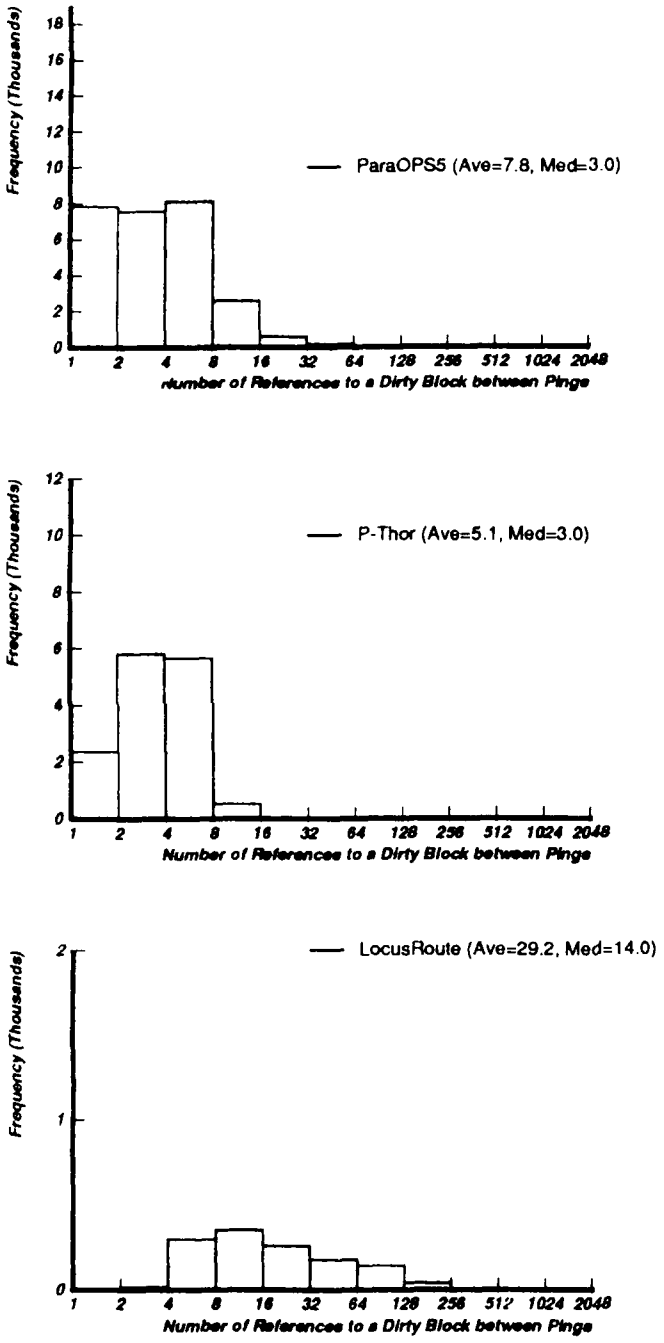

Figure 3: Distribution of the number of references to a block before a pinging reference to the same block, given that at least one reference was a write. Only *shared data* references of *user* are included. Block size is 16 bytes.

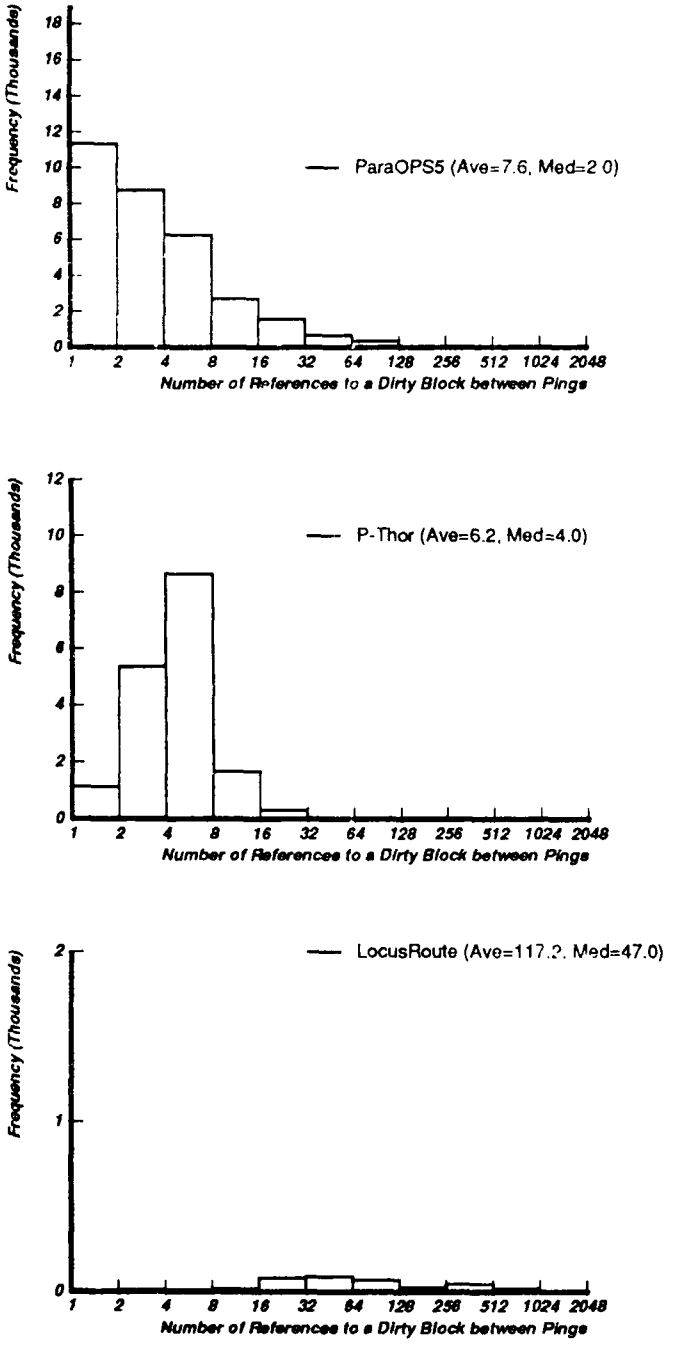

Figure 4: Distribution of the number of references to a block before a pinging reference to the same block, given that at least one reference was a write. Only *shared data* references of *user* are included. Block size is 64 bytes.

4.4 Cache Consistency Implications

Processor locality impacts the performance of cache coherence schemes. We examined the performance of several cache coherence schemes through simulation with the ATUM traces for various block sizes and used our notions of processor locality to gain insight into their behavior. Our findings are summarized here. We also discuss some of the limitations of our definitions of processor locality, and suggest modified definitions for use in specific applications.

Of the several cache coherence schemes proposed in the literature (e.g., [16, 14, 12, 17, 13]), we consider a representative each from the write-through with invalidate, write-back with invalidate, and write-back with update classes of cache coherence schemes assuming a shared bus as the communications medium.[5] To help explain the various phenomena we observed, we use the data presented in earlier sections. As before we assume infinite caches, and unless otherwise stated, block size is one word (or four bytes).

The *write-through with invalidate* scheme (WTI) is commonly used in low-end commercial multiprocessors. In this scheme, every write from a processor accesses the bus both to update main memory and to invalidate that location in other caches.

Examples of *write-back with invalidate* schemes include Goodman's write-once [16], Rudolph and Segall's scheme [17], Berkeley Ownership [14], and the directory scheme [20]. We consider write-once (denoted WBI) as the second scheme in this paper. In this scheme, the first write to a location uses the bus to update main memory and to invalidate that location in other caches. Subsequent writes to that location by the same processor do not result in any bus traffic, as that location is now owned locally.

Write-back with update schemes include Dragon [12] and Firefly [13]. We use Dragon as the third scheme, and denote it WBU. In the Dragon scheme, all writes to a shared location (a location present in multiple caches) result in a bus access to update the value of that location in other caches. For non-shared locations, the cache acts like a regular uniprocessor write-back cache.

We evaluated the performance of the above three cache coherence schemes in terms of the *bus transactions* generated. A bus transaction is generated on block transfers due to misses, invalidations, or updates. Because of our interest in characteristics of shared references, we only include cpu-shared user data references for ParaOPS5, P-Thor, and LocusRoute. Because caches are infinite, a data item brought into the cache remains there until invalidated

Before we discuss our results, we examine how we might choose an appropriate definition of processor locality for a given application. Recall the three variations of processor locality in Section 4.2. The first form simply counts the number of clinging references to a block ($r_c^*\ w_c^*\ r/w_p$). In other words, we use the average number of repeat references by a processor to a given block of data. The second form counts the number of clinging references to a block for those runs that included at least one write reference ($r_c^*\ w_c^+\ r/w_p$). Figures 2 through 4 plotted distributions using this second form. The third form counts just the number of writes in sequences of the second form ($w_c^+\ r/w_p$). Eggers and Katz define and use the same notion in their evaluation of cache coherence schemes in [5].

The first form is useful in analyzing cache coherence schemes that allow only one cached copy

[5] While a detailed analysis of the numerous cache consistency schemes proposed in the literature would be interesting, it is beyond the scope of this paper. Instead, see the simulation study of several cache coherence schemes by Archibald and Baer [18], and more recently, the simulation study using real address traces by Eggers and Katz [19].

of a block (e.g., the Dir_1NB scheme [8]). The second form is useful in examining ownership based protocols, where a writer must first become the sole owner of a block before proceeding with the write. The third form is necessary to distinguish between invalidating and updating (or write-through) protocols.

We first compared the performance of WTI, WBI, and WBU for 4-byte blocks using form three of processor locality. Comparing the number of transactions, we saw that the WTI scheme was worse than both WBI and WBU. WTI looses to WBI because of the processor locality displayed by write references. While every write generates bus traffic in WTI, clinging write references do not cause bus traffic in WBI. In fact, recall from Section 4.2, that on average there were 1.6, 1.7, and 1.2 writes in a sequence of clings before a ping for ParaOPS5, P-Thor, and LocusRoute respectively. Based on these numbers we verified that the greatest savings between WTI and WBI to be for P-Thor, next greatest for ParaOPS5, and least for LocusRoute. For example, WBI in P-Thor saves 49% bus transactions over WTI, ParaOPS5 saves 31%, and LocusRoute saves 11%.

Comparing WTI and WBU, both schemes generate an update transaction for every write to a shared location. However, WBU saves about 25% updates because before the point that a location becomes shared (a second processor requests it), only the first read or write produces a bus transaction. WBU also has fewer block transfers because, unlike WTI, it never invalidates a location from a cache.

Let us now compare WBI and WBU. WBI, in general, will be superior to WBU if there were a large number of clinging writes to an object before a ping. This is because, WBI does not produce bus traffic after the first write in a sequence of clinging writes. Again, recall from Section 4.2 that on average there are 1.6, 1.7, and 1.2 clinging writes for ParaOPS5, P-Thor, and LocusRoute respectively. Thus WBI has the greatest chance to win over WBU for P-Thor, next for ParaOPS5, and least for LocusRoute, which is borne out by simulations. WBI wins over WBU by 28% for P-Thor, by 3% for ParaOPS5, and loses by 21% for LocusRoute.

Dividing the total number of bus transactions generated by all three programs for the WBI scheme (161.6K) by the total number of references that resulted in these transactions (1168.7K), we see that there are approximately 0.138 bus transactions generated per reference. This number appears quite large given infinite caches, and there are two reasons for this. First, this data represents only cpu-shared user data references, which show poor processor locality as in Figure 2, or equivalently, which display a high temporal locality of pinging references. Consequently they do not benefit much from the read-sharing allowed by the WBI scheme. If one includes both user and OS references, and both data and instructions, then the number of transactions per reference falls to 0.031, which is much better. This reduction is primarily due to the large number of read-shared references generated by instruction fetches. When the block size is increased from 4 to 16 bytes, the number of transactions per reference further drops down to 0.016, primarily due to the high spatial locality of instruction fetch references.

We then examined the bus transactions generated by WBI as the block size is increased to study the spatial locality characteristics of cpu-shared user data references. For this analysis the second form of processor locality is relevant because once a block is read, a transaction takes place only on a pinging reference -- on a pinging read the block must be written back to memory, while on a write the block must be invalidated.

We observed that the measure of processor locality using the second form correctly predicts the trends in ParaOPS5 and LocusRoute. For example, the transaction rate in ParaOPS5 decreases when the block size is changed from 4 to 16 bytes, and the number of transactions

increases when the block size is further changed to 64 bytes. A corresponding increasing trend is observed in the second form of the processor locality parameter (see Figures 2 through 4).

A different trend is observed in LocusRoute as the block size is increased. The transaction rate decreases as we go from 4 to 16 to 64 bytes, a corresponding improving trend is displayed by the processor locality parameter for LocusRoute as the block size is increased.

The trends in P-Thor, however, did not match completely. A possible reason for the disagreement we observed is that the second form of processor locality as defined by us corresponds most closely to a protocol that invalidates a currently dirty copy of a block in a cache on a pinging read rather than just performing the writeback and making it clean. If a more accurate processor locality metric for analyzing performance of ownership protocols that clean rather than invalidate is desired, one can measure the average length of sequences of references to a block of data by a given processor, terminating the sequences only on pinging writes. This form of processor locality is denoted $r_c^* \; w_c^+ \; w_p$. The important observation is that the notion of pings and clings make it possible to customize the processor locality definition to suit a particular application.

5 Summary and Conclusions

We have characterized locality in memory reference patterns of shared-memory multiprocessors. Our data is based on traces obtained for three applications from a 4-processor VAX 8350 using the ATUM address tracing technique. About one-fifth of the references in the traces are to shared objects.

Shared references display a significant amount of temporal locality, but only a moderate amount of processor locality for both read and write references. For example, the average number of reads and writes to a write-shared block before a remote reference (a ping, which may possibly invalidate the data) are 4 and 2 respectively. Nevertheless, caching shared data is still highly useful because of the significant amount of read sharing. Although the average number of writes to a block before a remote reference is just 2, we observed a high variance in the length of write sequences. We believe that the use of hybrid updating and invalidating schemes, such as updating for synchronization objects and invalidating for others, or a dynamic competitive cache management strategy will prove useful in such environments.

The locality characterization of the shared-memory reference patterns also yields insight on how various cache consistency schemes will perform. We analyzed three classes of cache consistency schemes—write-through with invalidate (WTI), write-back with invalidate (WBI), and write-back with update (WBU). For shared data references, WTI performs worse than both WBI and WBU as it uses the bus on every write. Comparing WBI and WBU, the former seems to have an edge for 4-byte blocks, while WBU does better for 16-byte and 64-byte blocks. The processor locality parameter shows that blocks larger than 16 bytes in P-Thor and ParaOPS5 cause a degradation in processor locality, and thus the total bus traffic increases rapidly with increasing block size. The WBU scheme is less influenced by the block size than WBI and WTI because it always uses single word updates. Consequently, for large block sizes, WBU performs better than WBI and WTI for all three programs.

6 Acknowledgments

We thank Roberto Bisiani and the Speech Group at CMU for letting us use their VAX 8350 for collecting the traces used in this study. Dick Sites at Digital Equipment Corporation, Hudson, made the ATUM microcode available for our use. Larry Soule and Helen Davis at Stanford helped with the P-Thor program and Jonathan Rose with LocusRoute. Discussions with Susan Owicki, Susan Eggers, Mark Horowitz, John Hennessy, and Rich Simoni are also gratefully acknowledged. The research reported in this paper was funded by DARPA contracts # MDA903-83-C-0335 and # N00014-87-K-0825. Anoop Gupta is also supported by a faculty development award from DEC.

References

[1] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[2] J. R. Spirn. *Program Behavior: Models and Measurements. Operating and Programming Systems Series*, Elsevier, New York, 1977.

[3] F. Darema-Rogers, G. F. Pfister, and K. So. Memory Access Patterns of Parallel Scientific Programs. In *Proceedings of ACM SIGMETRICS 1987*, pages 46–58, May 1987.

[4] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proceedings of ACM SIGMETRICS 1988*, May 1988.

[5] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[6] Richard L. Sites and Anant Agarwal. Multiprocessor Cache Analysis using ATUM. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 186–195, IEEE, New York, June 1988.

[7] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[8] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[9] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, IEEE, New York, June 1986.

[10] Anoop Gupta, Charles Forgy, and Robert Wedig. Parallel Architectures and Algorithms for Rule-Based Systems. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, IEEE, New York, June 1986.

[11] Jonathan Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Design Automation Conference*, pages 189–195, June 1988.

[12] E. McCreight. *The Dragon Computer System: An Early Overview.* Technical Report, Xerox Corp., September 1984.

[13] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[14] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, IEEE, New York, June 1985.

[15] Anna Karlin, Mark Manasse, Larry Rudolph, and Daniel Sleator. *Competitive Snoopy Caching.* Technical Report CMU-CS-86-164, Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 1986. Preliminary version appeared in 27th FOCS, 1986.

[16] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, IEEE, New York, June 1983.

[17] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340–347, IEEE, New York, June 1985.

[18] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[19] S. J. Eggers and R. H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, IEEE, New York, June 1989. To appear.

[20] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy